applications using the engine that should know the details of the interface (methods & properties) exposed by the engine and the associated algorithms. Once the designer has constructed the simulation model (Excel Spreadsheet) and configured all the inputs, outputs & lists, he is ready to test using the test bench included in the ICA Utilities (refer to ICA Utilities documentation). The developer, in turn, needs to implement the calls to the simulation engine in the GBS application he's building. The following list identifies the files that need to be included in the Visual Basic project to use the simulation workbench :

wSimEng.cls	Simulation Engine class
wSimEng.bas	Simulation Engine module (this module was introduced only for speed purposes because all the code should theoretically be encapsulated in the class)
wConst.bas	Intelligent Coaching Agent constant declaration
wDeclare.bas	Intelligent Coaching Agent DLL interface
wica.cls	Intelligent Coaching Agent class
wica.bas	Intelligent Coaching Agent module (this module was introduced only for speed purposes because all the code should theoretically be encapsulated in the class)

To have a working simulation, a developer places code in different strategic areas or stages of the application. There's the Initial stage that occurs when the form containing the simulation front-end loads. This is when the simulation model is initialized. There's the Modification stages that take place when the user makes changes to the front-end that impacts the simulation model. This is when the ICA is notified of what's happening. There's the Feedback stage when the user requests information on the work done so far. This is when the simulation notifies the ICA of all output changes. Finally, there's the Final stage when the simulation front-end unloads. This is when the simulation is saved to disk.

The different stages of creating a simulation, including the Visual Basic code involved, are presented below. Initial stage; **1. Creating the ICA & the simulation engine object:** Code: Set moSimEngine = New classSimEngine; Set moICA = New classICA; Description: The first step in using the simulation engine is to create an instance of the class classSimEngine and also an instance of the class classICA. Note that the engine and ICA should be module level object "mo" variables. **2. Loading the simulation**; Code: IRet = moSimEngine.OpenSimulation(App.Path & DIR_DATA & FILE_SIMULATION, Me.bookSimulation); IRet = moSimEngine.LoadSimulation(mlICATaskID, App.Path & DIR_DATABASE & DB_SIMULATION, 1); Description: After the object creation, the OpenSimulation and LoadSimulation methods of the simulation engine object must be called. The OpenSimulation method reads the specified Excel 5.0 spreadsheet file into a spreadsheet control. The LoadSimulation method opens the simulation database and loads into memory a list of paths, a list of inputs, a list of outputs and a list of lists for the specific task. Every method of the simulation engine will return 0 if it completes successfully otherwise an appropriate error number is returned. **3. Initializing and loading the Intelligent Coaching Agent**; Code:IRet = moICA.Initialize(App.Path & "\" & App.EXEName & ".ini", App.Path & DIR_DATABASE, App.Path & DIR_ICADOC, App.Path & "\"); IRet = moICA.LoadTask(mlICATaskID, ICAStudentStartNew); Description: The simulation engine only works in conjunction with the ICA. The Initialize method of the ICA object reads the application .ini file and sets the Tutor32.dll appropriately. The LoadTask method tells the ICA (Tutor32.dll) to load the .tut document associated to a specific task in memory. From that point on, the ICA can receive notifications. Note: The .tut document contains all the element and feedback structure of a task. Ex: SourcePages, SourceItems, TargetPages, Targets, etc... **4. Restoring the simulation**; Code: <<Code to reset the simulation

when starting over>>; <<Code to load the controls on the simulation front-end>>; IRet = moSimEngine.RunInputs(sPaths, True); IRet = moSimEngine.RunOutputs(sPaths, True); IRet = moSimEngine.RunLists(sPaths, True); Call moICA.Submit(0); Call moICA.SetDirtyFlag(0, False); Description: Restoring the simulation involves many things: clearing all the inputs and lists when the user is starting over; loading the interface with data from the simulation model; invoking the RunInputs, RunOutputs and RunLists methods of the simulation engine object in order to bring the ICA to it's original state; calling the Submit method of the ICA object with zero as argument to trigger all the rules; calling the SetDirtyFlag of the ICA object with 0 and false as arguments in order to reset the user's session. Running inputs involves going through the list of TutorAware inputs and notifying the ICA of the SourceItemID, TargetID and Attribute value of every input. Running lists involves going through the list of TutorAware lists and notifying the ICA of the SourceItemID, TargetID and Attribute value of every item in every list. The TargetID is unique for every item in a list.

Running outputs involves going through the list of TutorAware outputs and notifying the ICA of the SourceItemID, TargetID and Attribute value of every output. Modification stage 1. **Reading inputs & outputs;** Code: Dim sDataArray(2) as string; Dim vAttribute as variant; Dim ISourceItemID as long; Dim ITargetID as long; IRet = moSimEngine.ReadReference("Distinct_Input", vAttribute, ISourceItemID, ITargetID, sDataArray)

Description: The ReadReference method of the simulation object will return the attribute value of the input or output referenced by name and optionally retrieve the SourceItemID, TargetID and related data. In the current example, the attribute value, the SourceItemID, the TargetID and 3 data cells will be retneved for the input named Distinct_Input.

Description: The simulation engine object provides basic functionality to manipulate lists.

The ListAdd method appends an item(SourceItemID, Attribute, Data array) to the list. Let's explain the algorithm. First, we find the top of the list using the list name. Then, we seek the first blank cell underneath the top cell. Once the destination is determine, the data is written to the appropriate cells and the ICA is notified of the change.The ListCount method returns the number of items in the specified list. The algorithm works exactly like the ListAdd method but returns the total number of items instead of inserting another element.The ListModify method replaces the specified item with the provided data. Let's explain the algorithm. First, we find the top of the list using the list name. Second, we calculate the row offset based on the item number specified. Then, the ICA is notified of the removal of the existing item. Finally, the data related to the new item is written to the appropriate cells and the ICA is notified of the change. The ListDelete method removes the specified item. The algorithm works exactly like the ListModify method but no new data is added and the cells (width of the list set by 'Total Columns') are deleted with the 'move cells up' parameter set to true. Keep this in mind, as designers often enter the wrong number of columns in the Total Columns parameter. When they overestimate the Total Columns, ListDelete will modify portions of the neighboring list, which leads to erratic behavior when that list is displayed.

SYSTEM DYNAMICS IN ACCORDANCE WITH A PREFERRED EMBODIMENT

To use system dynamics models in the architecture, an engine had to be created that would translate student work into parameters for these models. A complex system dynamics model to interact with an existing simulation architecture is discussed below. The system dynamics model provides the following capabilities.Allow designers to build and test their system dynamics models and ICA feedback before the real interface is built.Reduce the programming complexity of the activities.Centralize the interactions with the system dynamics models.System Dynamics Engine As with the simulation engine, the designer models the task that he/she wants a student to accomplish using a Microsoft Excel spreadsheet. Here, however, the designer also creates a system dynamics model (described later). The system dynamics engine will read all of the significant cells within the simulation

model (Excel) and pass these values to the system dynamics model and the ICA. After the system dynamics model runs the information, the output values are read by the engine and then passed to the simulation model and the ICA.

Figure **27** is a block diagram presenting the detailed architecture of a system dynamics model in accordance with a preferred embodiment. Once the simulation model, system dynamics model and feedback are completely tested by designers, developers can incorporate the spreadsheet in a graphical user interface, e.g., Visual Basic as a development platform. Figure **27** illustrates that when a student completes an activity, the values are passed to the system dynamics engine where the values are then passed to the system dynamics model (as an input), written to the simulation model and submitted to the ICA. When the system dynamics model is played, the outputs are pulled by the engine and then passed to the simulation model and the ICA. Note that the simulation model can analyze the output from the system dynamics model and pass the results of this analysis to the ICA as well. The simulation model can then be read for the output values and used to update on-screen activity controls (such as graphs or reports).It is very important that all modifications that the ICA and system dynamics model need to know about go through the engine because only the engine knows how to call these objects. This significantly reduces the skill level required from programmers, and greatly reduces the time required to program each task. In addition, the end-product is less prone to bugs, because the model and tutor management will be centralized. If there is a problem, only one section of code needs to be checked. Finally, since the engine loads the data from the spreadsheet. the chance of data inconsistency between the ICA, the system dynamics model and the application is insignificant.

The system dynamics model generates simulation results over time, based on relationships between the parameters passed into it and other variables in the system A system dynamics object is used to integrate with Visual Basic and the spreadsheet object. The object includes logic that controls the time periods as well as read and write parameters to the system dynamics model. With Visual Basic, we can pass these parameters to and from the model via the values in the simulation object. The system dynamics object also controls the execution of the system dynamics model. What this means is that after all of the parameter inputs are passed to the system dynamics model, the engine can run the model to get the parameter outputs. The system dynamics object allows for the system dynamics models to execute one step at a time, all at once, or any fixed number of time periods. When the system dynamics model runs, each step of the parameter input and parameter output data is written to a 'backup' sheet for two reasons. First, the range of data that is received over time (the model playing multiple times) can be used to create trend graphs or used to calculate statistical values. Second, the system dynamics model can be restarted and this audit trail of data can be transmitted into the model up to a specific point in time. What this means is that the engine can be used to play a simulation back in time. When any event occurs within the system dynamics engine, a log is created that tells the designers what values are passed to the simulation model, system dynamics model and ICA as well as the current time and the event that occurred. The log is called "SysDyn.log" and is created in the same location as the application using the engine. As with the spreadsheet object, the system dynamics object allows a large amount of the calculations to occur in the system dynamics model and not in the activity code, again placing more control with the activity designers. Model objects are used to configure the system dynamics models with regard to the time periods played. Models are what the parameter inputs and parameter outputs (discussed later) relate to, so these must be created first. Every model has the following application programming interface:

Field Name	Data Type	Description
ModelID	Long	Primary Key for the table